pyCatalstReader: Extracting Text and Tokenization of Technical Catalysis Science Papers

Giordanno Castro Garcia

M.S. Computer Science, University of Kansas, 2021

Submitted to the graduate degree program in Department of Electrical Engineering and Computer Science and the Graduate Faculty of the University of Kansas in partial fulfillment of the requirements for the degree of Masters of Science.

Committee members

Dr. Michael S. Branicky, Chairperson

Dr. Fengjun Li, Member

Dr. Bo Luo, Member

Dr. Kevin Leonard, Member

Date defended: December 08, 2021

The Dissertation Committee for Giordanno Castro Garcia certifies
that this is the approved version of the following dissertation :

pyCatalstReader: Extracting Text and Tokenization of Technical Catalysis Science Papers

Dr. Michael S. Branicky, Chairperson

Date approved: December 08, 2021

Abstract

Catalysts are an essential and ubiquitous component of our modern life, from empowering our agriculture to reducing toxic emissions. There is a constant need for more and better catalysts. The catalysis research literature is immense, growing, and scattered. Natural Language Processing (NLP), a sub-field of Machine Learning (ML), offers a potential solution to automatically make full use of all this valuable information and speed innovation. Even though NLP has made much progress in the analysis of everyday text, its application in more technical text has not been as successful. Specifically, there are even a dearth of tools that can appropriately extract text from the PDF files of research articles, which are the most common format used in the catalyst field. Therefore, this project aims to define a tool that can extract text from PDF files of catalysis science articles, which is prerequisite to applying NLP and ML tools. We also explore the first stage of the NLP pipeline, tokenization, by objectively comparing different tokenizers for catalysis science articles.

Acknowledgements

I would like to thank my advisor, Dr. Branicky, and my committee for the help when preparing this document. Also, I would like to thank my teammates (Emily, Joe and Christian), my parents, brother, Eiman and my extremely white schnauzer: Clark. Your support and will to listen to me was infinitely helpful these past years.

Contents

List of Figures

Chapter 1

Introduction

We live in an age where Machine Learning (ML) has affected several areas of our life. From autonomous drones that get the best shot of its user to software that can seamlessly remove objects from a picture, researchers keep finding new fields where they can apply ML. Natural Language Processing (NLP) is a "theoretically motivated range of computational techniques for analyzing and representing naturally occurring texts" [6]. NLP has several applications, the most popular being sentiment analysis where a system can accurately guess whether a section of text has a positive or negative sentiment attached to it for example. However, NLP offers plenty of opportunities as it allows us to automate the analysis of text.

Currently, the field of catalyst development has a problem that could have NLP as its potential solution. Catalysts accelerate the rate at which a chemical reaction happen [13]. Our modern lives highly depend on catalysts as they are used in the agriculture industry, automotive industry, and in many other situations. Due to their ubiquitous usage, researchers are in a never-ending process to develop more efficient catalysts. This situation has lead to an overflow of publications that researchers often have to ignore as they cannot physically read them all. However, the problem this field faces lies in the fact that due to this constraint, researchers often ignore information that could lead to the next big discovery. As previously mentioned, NLP is concerned with helping computers understand human language. Therefore, NLP could help catalyst researchers if it could offer a system that could automate the extraction of relevant information out of research papers. Figure 1.1 shows the stages that we would need to follow to go from research papers in HTML/PDF format to our final goal, which in this case is the discovery of new catalysts.

The NLP processing step tends to be defined as a pipeline of processes where each stage gener-

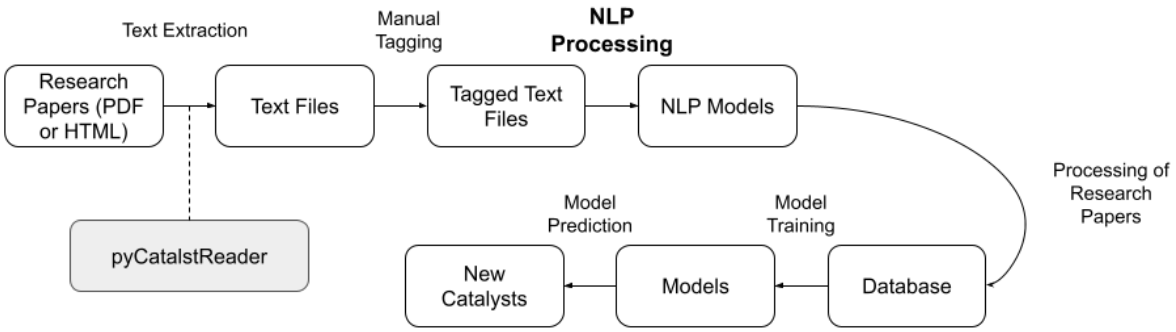

Figure 1.1: Expected steps to go from text extraction to predicting catalysts.

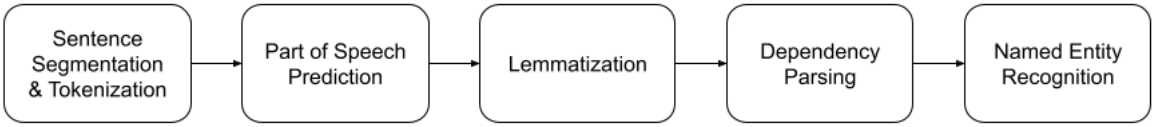

Figure 1.2: Stages of a general NLP pipeline.

ates a result that feeds the next stage. Figure 1.2 shows one of the possible pipelines that includes five different steps: Sentence Segmentation and Tokenization, Part of Speech Prediction, Lemmatization, Dependency Parsing, and Named Entity Recognition. For our specific goal of helping researchers explore papers with relevant information, Named Entity Recognition (NER) seems like a possible solution. NER could identify relevant entities (catalysts, reactions, units, etc) in the piece of text that we provide to it. However, as Figure 1.2 shows, NER needs several stages of pre-processing before it can identify those entities.

Therefore, our goal is to cover two topics: the accurate extraction of text out of PDF files as well as the identification of an appropriate Tokenization system. Even though both of these goals already have plenty explored solutions, the catalyst vocabulary adds an extra layer of complication that we will need to navigate around to come up with an appropriate solution.

Chapter 2

Objective

The objective for this work is to present a Python package that can automate the extraction of text out of a PDF. Specifically this package needs to do the following:

- Extract text correctly, even if the paper is formatted with two columns.
- Identify the headers within a research paper.
- Trim margins from the papers to get rid of footer text.
- Remove images to remove text that doesn't make sense.
- Ability to delete certain sections from a paper.
- Providing an easy-to-use API for developers that is easily accessible.
- Offer both individual file as well as batch versions of this tool.

This tool is aimed towards power-users who may not have extensive programming knowledge. Therefore, we tried to simplify the interface as much as possible while still offering a wide set of tools. The hope is that this package will make it easier for NLP researchers to extract text and generate a corpus that fits the needs of their research. Due to the multidisciplinary nature of this objective, I worked in collaboration with Joseph Karnes, Emily Mikeska, and Christian Nilles, all of them Chemical Engineering PhD students as part of an NSF NRT grant [10]. Figure 2.1 show the stages that pyCatalstReader goes through to extract structured text out of a PDF file.

Once we finished the text extraction tool, I moved on to the next step of the NLP pipeline which traditionally is tokenization. Tokenization of chemistry entities proved to be quite complex as they

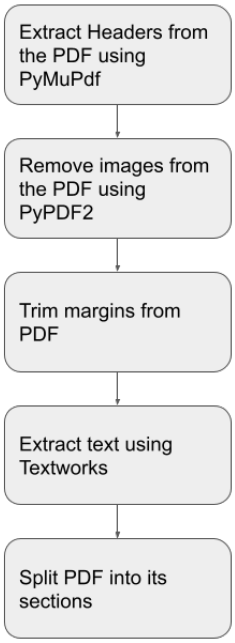

Figure 2.1: Stages in pyCatalstReader to extract text out of a PDF file.

tend to have a series of characters that normally would be considered separators in other fields such as '-', '+', super- and sub-scripts, etc. However, we found a decent amount of tokenizers available online. Therefore, I focused on developing a series of metrics that could help us choose the tokenizer that best fitted our needs.

Chapter 3

Motivation

The motivation for this work is to try to develop a tool that will enable the extraction of content out of chemistry related research papers. There are plenty of opportunities to revolutionize the field of catalysts through the application of AI. New research papers get published constantly with new information that can help the development of the next big discovery. However, due to the raw volume of all of this new pieces of research, it is impossible for a single researcher to parse most of them. Therefore, there is a need to develop a system that can automatically extract relevant information out of them or maybe summarize them or maybe highlight the most relevant papers for a researcher in the field of chemistry/chemical engineering.

However, as with most AI-based solutions, we need to train a model that will perform the task that we want to achieve (whether it is recommendation or classification or entity extraction). To train this model we need lots of training data which in this case we do not have. There is a lack of a corpus for the catalyst field that can be used to train such models. To complicate things even further, most of papers in the catalyst field are released in PDF format. As I will show later, PDFs are a file format that is tricky to deal with as different journals have different formats. Things such as the double column, headers and other items can complicate the extraction of text out of them.

While there are tools that can extract text out of PDFs, no tool can cover all of the cases that catalyst researchers want to cover. Therefore, this project aims to build a tool that is capable of extracting text out of PDFs. However, since this product aims to be used by researchers that do not have wide programming experience, we attempted to make it as simple as possible to use. In fact, we tried to pay close attention to the simplicity of our package and hope that others will find it as intuitive as we do.

Once we were able to extract text using our own package, we aimed to focus on the next step of the PDF pipeline: tokenization. Tokenization is the process of splitting a piece of text into basic units called tokens. Intuitively, we could say that a token is a piece of text split by white-spaces. However, there are some cases in which this assumption does not hold true. For example, in the English language, we have several instances of compound words such as long-term that can be split into 2 individual tokens: long and term. Several of these rules do not apply to the chemistry field, where entities can be connected by dashes or signs such as +, -, etc. Even though it is easy to assume that tokenization is a trivial process, in fact it is an ambiguous one. We did not aim to develop our own tokenizer since this would have demanded resources that we did not have available. Fortunately, we found several tokenizer options available online that were aimed towards the chemistry field. However, we did not have any way to objectively compare them. Therefore, I developed a series of metrics that could compare each tokenizer with a manually-tokenized list so that we could select the one that performed the best for our needs.

Chapter 4

Background

As explained before, our long-term goal is to develop a framework that can identify chemical entities in research papers focused on homogeneous chemistry. Others have attempted to achieve this goal before. We have found two such packages:

- ChemDataExtractor [14]
- Chemlistem [3]

ChemDataExtractor is defined as a "toolkit for the automated extraction of chemical entities and their associated properties, measurements, and relationships from scientific documents..."[14]. Whereas Chemlistem is defined as a collection of systems that perform "chemical named entity recognition" [3]. Both packages are focused on extracting/recognizing chemical entities within a piece of text. However, both of them have different approaches to solve this problem. While ChemDataExtractor uses Conditional Random Fields (CRF), Chemlistem provides three different systems: one that also uses CRFs, another one that uses Long Short Term Memory (LSTM) layers, and finally a hybrid system that combines both of these. However, for this report we will focus on the beginning stages of both of these systems.

ChemDataExtractor (CDE) has its own system that allows it to convert files (PDF, HTML, XML) to text. However, Chemlistem doesn't have a way to extract text out of a file, it expects a string. Therefore, we focused on CDE for our early tests as it provided an available pipeline from research paper file to chemical entities. However, our early tests showed that while CDE performs adequately on generic chemical entities, it does not perform well with the papers relevant to us. Chemlistem's three different systems also do not perform well on our papers.

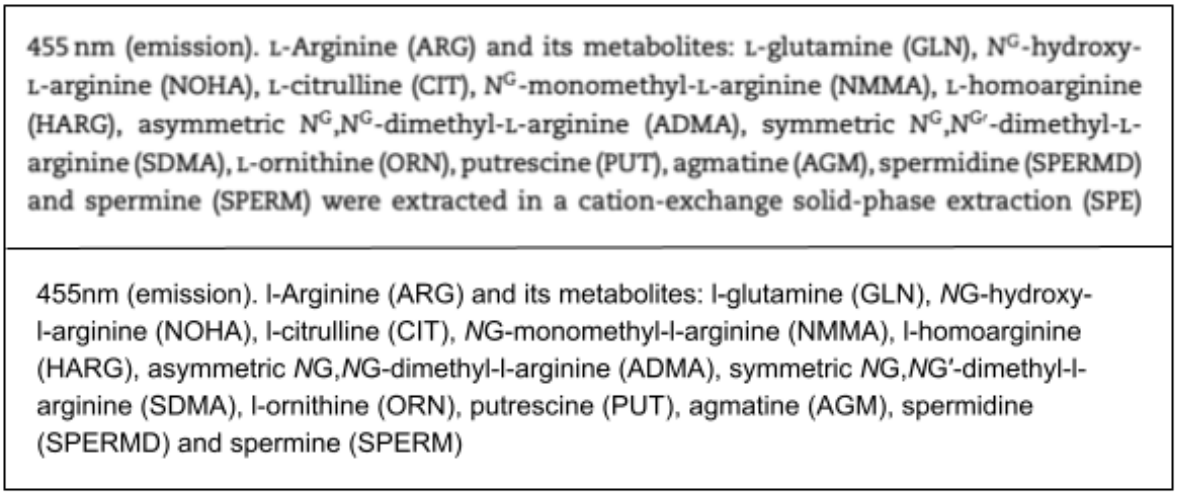

Figure 4.1: On the top the original text and on the bottom the extracted text using CDE.

We identified the following issues in both systems that we tested:

- Lack of proper support for double-column formatted text.
- Lack of support for super- and sub-scripts.
- Inability to identify and discard sections from PDF text.
- Forced inclusion of out-of-context text from images in PDF.
- In some cases, bad tokenization or bad sentence segmentation.

As mentioned before, CDE offers a method to extract text out of files (PDF, HTML, XML). The majority of the papers relevant to our topic come in a PDF format so we focused on testing CDE's PDF text extraction method and found it to lack some features. The most noticeable is the inability to identify super- and sub-scripts. Figure 4.1 shows a comparison between the original text in the PDF file and the text that CDE extracts. Entities like "N^G-hydroxy-L-arginine" are extracted as "NG-hydroxy-L-arginine". In the field of chemistry, subscripts and superscripts are crucial to distinguish between chemical entities. For example $H_2O^+(^2A_1)$ refers to a water molecule ion in the doublet-A-one state. However, according to CDE, this text would be H2O+(2A1), which is

not a valid chemical entity. Another example are ions such as tetrafluoroborate such as BF_4^-, CDE would parse this as BF4-, which again is not a valid chemical entity.

Also, CDE extracts the text line by line, meaning that the text it returns in Python has newline characters at the end of each line. In some cases, words are split using a dash, which can make it difficult to understand what word the text is referring to. Figure 4.1 shows that the entity N^G-hydroxy-L-arginine is split in two lines. Therefore, we need a way to combine all of the lines together and also be able to put words together that are split into two.

At this point, we identified the need to develop our own PDF text extraction tool. However, there are tools that already claim to do this, such as:

- pdftotree
- poppler-utils
- pdfminer
- TextWorks
- PyMuPDF

pdftotree is a Python package that aims to build an HTML tree from a PDF file [4]. While this package accurately recognizes characters, it omits key features, such as extracting the text in the right column order and identifying sub- and super-scripts. Additionally, this package adds a lot of metadata that we would have to clean to obtain the text. poppler-utils is another Python package that can split the text if the paper has a double column format [9]. However, it does not capture sub- and super-script characters, and it can also read the text out of order. pdfminer is another Python package that does a good job at reading the text in the right order in double column formatted papers [12]. Unfortunately, it does not identify sub- and super-scripts. TextWorks is a command line tool that extracts PDF text into JSON files [1]. Some of its key advantages are its ability to recognize sub- and superscripts, as well as its correct handling of double column formatted text. However, we would need to reconstruct the text based on the JSON file that TextWorks returns.

PyMuPDF is a pdf toolkit can extract text out of PDFs [8]. Its key feature is its ability to extract font information out of PDF text. However, it does not recognize sub- and super-script text.

Chapter 5

Methodology

In addition to all my technical contributions discussed elsewhere, as the only Computer Science member, I led my team's efforts when developing the package. I attempted to lead my team and the general NRT group towards the use of traditional programming tools such as Git and Slack.

Particularly, I pushed towards the use of a version control system such as Git along with Github. I created the Internet-of-Catalysis-KU organization as well was the individual repositories for each team. The NRT project had three teams:

- Team LA
- Team Metha
- Team CO2

I created a team in Github for each team and added the corresponding members to each one. Then, I moved on to create the individual repositories for each team and established a hierarchy of users for each repository. Figure 5.1 shows an example of how I established these roles. I gave access to all teams to visualize all repositories, but I only gave access to Maintain (modify, commit and approve pull requests) to the team that owned the repository. Finally, I added one administrator per team, in my team's scenario that person was myself. The goal of this structure was to promote the exchange of ideas between teams by allowing each team to look at each other's code. However, to keep everybody's work safe, only the respective team had the rights to modify the files in their repository.

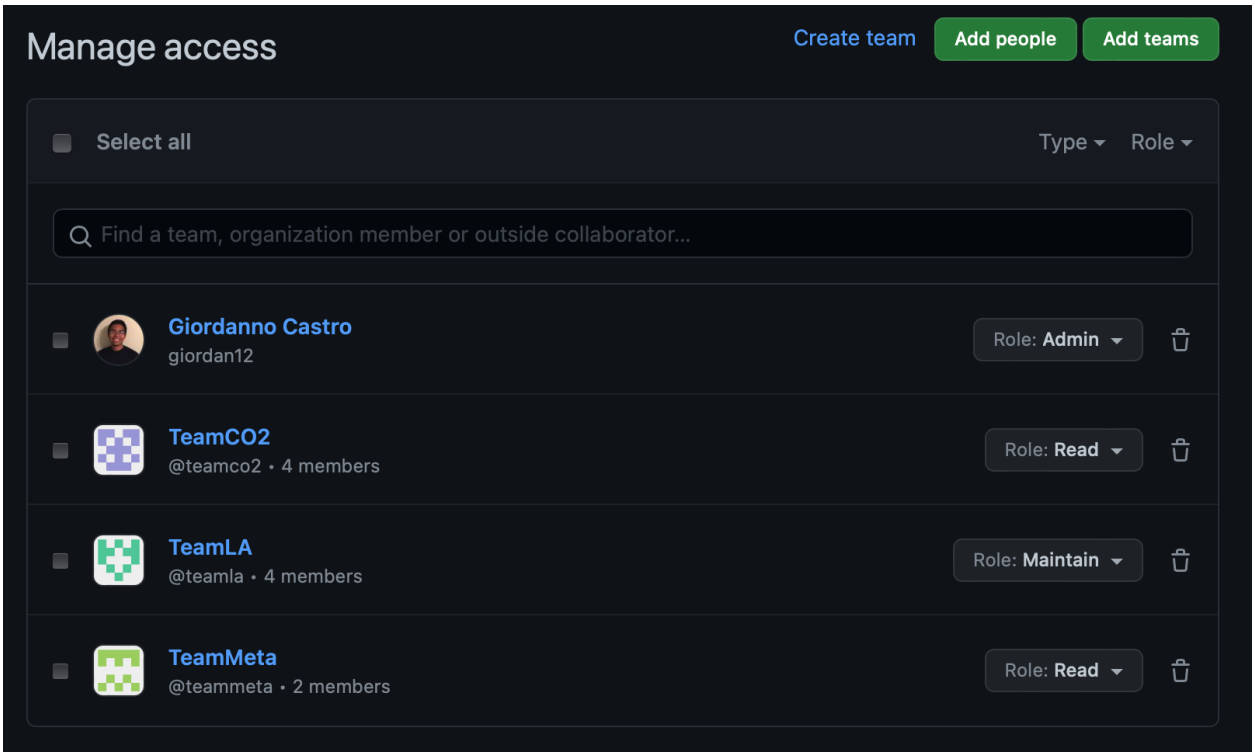

Figure 5.1: Example of the roles in Team LA's repository

With the goal to ease the transition of the other members of the NRT group into Github, I prepared a presentation. In this presentation, I covered basic Git concepts such as branches, commits, pull requests and merges. Also, I collected a series of links that I felt might be useful for people to expand their knowledge in Github. Finally, I met with a chemistry student (Andrew Jenny) to make sure that my teaching plan was accessible to people without previous knowledge in the topic and also to come up with a strategy to reduce the number of merge conflicts in Git.

Also, I established the following rules:

- Each member will have his/her individual branch and there will be a development and master branch.
- The development branch will serve as an integration branch to test individual commits together. The master branch will hold final tested changes to the codebase.
- Each person should work on a different file to avoid merge conflicts.

- Teams should aim to split their code in at least 3 different files (2 for corpus processing and 1 for NLP tasks).

The process in Slack was similar, where I created the organization in Slack and created individual channels in Slack for each team. I promoted Slack as a tool to exchange files and share snippets of code between team members.

Chapter 6

Presentation of Work

As explained in the previous sections, the end goal is to develop an NER model that can accurately extract relevant chemical entities from research papers. In our specific case, we focused on reactions that involve homogeneous catalysts.

Originally, we started by researching the current availability of tools that did something similar. We found 2 of them: ChemDataExtractor and ChemListem. Both of these Python packages claimed to be able to extract chemical entities out of research papers in different formats. However, ChemDataExtractor was the one that seemed more closely related to what we wanted to achieve so we investigated it more in depth.

Specifically, ChemDataExtractor claims that it can extract chemical entities out of HTML, XML, and PDF file formats. Its website also mentions that it utilizes a full NLP pipeline, which allows it to perform the chemical named entity recognition. ChemDataExtractor provided a good foundation for understanding what we wanted to achieve with our product. However, we noted that ChemDataExtractor did not perform as well as we hoped on the files that we provided. Therefore, we focused on the search for other frameworks that could perform something similar.

During this search we stumbled upon the Stanford NLP Group. At the moment of our search, this group had available a Python package that could interface with its Java CoreNLP library. Specifically, this Java library made available a toolkit for performing NLP analysis on text. This Python library, later renamed Stanza, allowed us to train our own models to perform NER. We went through the initial steps of first getting accustomed with the library and its options, and then we moved on to training our first simple models. These models performed relatively well, at that moment we decided to try to train larger models that could identify more complex chemical

entities.

It is at that moment that we encountered a traditional challenge in the AI world: lack of appropriate data to train our models on. So, our priority shifted towards acquiring enough data to train our models. Unfortunately, we didn't find any corpus relevant for our field. Therefore, we had to start by first collecting the data ourselves to move on to train a model that could perform the task that we wanted it to do.

We started by identifying the file format that contained most of our data. For our field, the most common file is PDF. Therefore, our goal became to extract text out of PDF files. ChemDataExtractor claimed to be able to extract chemical entities out of research papers in PDF format. Therefore, somewhere in its code structure, there had to be a method to extract the text out of the PDF files. Fortunately, ChemDataExtractor is shaped in such a way that it allows easy usage of its internal libraries. Unfortunately, its built-in text extraction method was not accurate and did not perform as well as we had hoped. We moved on to test different PDF-to-text libraries, but none of them performed as well as we hoped either. We identified three big issues with the packages that we tested: lack of support for super- and sub-scripts, lack of support for double-column formatted text, inability to identify headers in text.

Lack of support for super- and sub-scripts: Chemical text uses plenty of super- and sub-scripts. Some of the packages that we tested were not able to identify these character styles at all, returning them just as regular text. In chemistry, the difference between super- and sub-script characters can mean completely different entities. Therefore, we wanted to find a package that could accurately identify these character styles and appropriately convey them in their text version. The package that we found called PyPDF2 is able to do so by representing characters in sub-script using the format "_{}" and super-script by "+{}".

Lack of support for double-column formatted text was a key issue for us as well since most of the papers that we analyzed have this specific format. Some of the packages that we tested could accurately extract the text, but they would combine the text from both columns into a single line. Therefore, the text would be disorganized and it would not make any sense at all. Fortunately,

PyPDF2 was able to appropriately extract text from the column on the left and then move on to the right column.

One of the key issues that we noted when we first began analyzing the papers was that certain sections would contain a lot of disorganized information. Take, for example, the References section. It will contain several titles of papers relevant to our topic. However, the titles themselves do not necessarily contain information relevant to the topic the current paper is discussing. Therefore, we wanted to come up with a way to discard certain sections of the papers. Furthermore, we wanted to offer the user the ability to select certain sections out of their PDF files when extracting the text out of it. We tested several packages, but we could not find one that could accurately identify and split the text based on the headers. One of the key issues with the PDF format is the lack of structure in the contents of the file. In contrast, HTML offers a lot of information about the structure of the contents using its tags. Sections are well delimited and also headers are appropriately identifiable. Graphs are contained within special tags, so it is easy to manipulate them as well. Unfortunately, PDF offers none of those benefits. We considered for a bit focusing only on research papers shared using HTML; however, we noticed that this would reduce the impact of our package greatly as the majority of papers are shared via PDF. Therefore, we focused on developing a system to identify the headers in a PDF file. We started by analyzing the way that humans identify headers. Normally, a header is identifiable because it has a different font than the majority of the text in the file. Therefore, we explored packages that would return font information of text in PDFs. The best one that we could find was PyMuPDF. We started by analyzing the different font sizes throughout the text using custom functions. We expected to obtain at most six different font sizes, but in our tests on different papers, we obtained 40 to 50 different fonts. We looked a bit deeper into why this was happening. We discovered that text in subscript or superscript returns different font sizes. Also, ligatures tend to have different font sizes. We also saw how paragraphs can also have different font sizes. This can be due to some encoding artifacts when generating the PDF, but also it may be due to the authors using different sizes to fit their content within a certain amount of pages. This situation made us discard our idea of using the second or third most

common font size to identify the headers in a paper. However, we did notice that papers have some common sections. For example, we noticed that acknowledgement(s) and reference(s) tend to show up in most papers. Therefore, our approach to identify headers involves finding either one of the common headers, extracting its font information and then using this information to match it with other text in the document with similar font information. We had to take into consideration special cases with headers that have more than one font (if it includes sub/super script) and headers that span multiple lines. However, after taking care of these scenarios, we have a system that works well in most cases. If the PDF does not have a header Acknowledgments(s) or Reference(s), the system will not work though. Also, the system is not able to recognize sub-headers. Normally, sub-headers have a font size that is between the size used for regular headers and the one used in the paragraphs. Since there is no sub-header that is used in every paper, we cannot uniformly determine the sub-headers font information, and thus we cannot match it with other sub-headers.

Another tool that we wanted to offer our users was the possibility of removing images from the PDF. In our early tests, plenty of times we noted that graphs in research papers tend to contain text that doesn't make much sense without the data visualizations. This text would, most of the time, affect the text in the surrounding paragraphs.

This leads to the limitations of our PDF package. In its current version, it is not able to extract semantic information from graphs. Also, we are not able to extract information from tables either.

The pipeline of our package is as follows:

- We remove the images using PyPDF2.
- We extract the headers using font information extracted through PyMuPDF.
- We crop the margins of the PDF to get rid of text that commonly comes on the sides.
- We finalize by extracting the text using TextWorks.

Initially, our tool was built as a command-line tool that could be invoked through a makefile. We implemented it this way because one of the packages that we used, TextWorks, only offers a

command line interface. Therefore, the simplest option was to invoke our other scripts using the command line as well. This tool would expect three folders: one holding the PDFs, one to hold the versions without images, and finally one to hold the extracted text for each one. Therefore, this system could be thought as batch processing over the files that were contained in the initial folder.

However, we have also developed a Python package that contains the same functionality as the command line tool that we explained before and adds some new functionalities. The main purpose behind this is to enable developers to extract text and process it from the same Python script. If we would have kept the command-line tool, a user would have to invoke the command-line tool in their terminal and then move on to a Python script where they would have to import the text file. Since the end user of our package will be people that do not have extensive programming experience, we wanted to keep the interface simple.

Our Python package is split into two sections:

- Individual File Reader
- Batch File Reader

6.1 Individual File Reader

The individual file reader is invoked by using the constructor PDFReader. In its parameters you can specify the path to the PDF, the publisher, and the path to the json file generated by TextWorks if for some reason the user wants to define this path separately. When the constructor gets called, the package will extract the headers and these will be stored in a headers class attribute.

As explained before, the way that the package identifies headers is by using PyMuPDF and extracting font information from headers that tend to be common in catalyst-related papers such as:

- "References" and its variants such as "Reference", "REFERENCES", "Reference.", "References."

- "Acknowledgments" and its variants such as "Acknowledgements", "ACKNOWLEDGMENTS", "ACKNOWLEDGEMENTS", "Acknowledgments."

While we have tried to identify as many variants of the two previous headers as possible, there is a chance we have missed one. Also, it could be that the paper the user inputs does not have one of these common headers. If such is the case, then the system will stop as we want to ensure that the user can get access to the text split by headers.

Then the user can use the extractText method that offers parameters to specify whether the user wants to remove images from the pdf, format the text using the recognized headers, crop the PDF margins, or ignore certain headers during the text extraction process. The first parameter (remove_images) uses a boolean to determine whether images should be removed from the PDF during the text extraction process. If this argument is set to True, the system will use PyPDF2 to remove the images from the PDF and store the image-less version of the file in another file with the suffix "_Imageless" appended to the original filename. The second argument (header_formatted) is used to specify whether the text should be formatted using the extracted headers. This will help readability of the text file, but will not have any major effect in the text itself. The third parameter (crop_pdf) is used to determine whether or not the system should crop the margins of the PDF. Some PDFs have text in the margins that is not directly related to the topic being discussed in the document. This text can get included in the extracted text, so we want to offer the user the ability to crop the margins of the PDFs and get rid of those files. Currently, we perform the cropping by identifying the publisher and date of publishing to determine the format that the paper follows. The publishers we currently support are:

- American Chemical Society (ACS)
- Royal Society of Chemistry (RSC)
- Wiley
- Elsevier BV

For the publisher Wiley, we support only the following journals:

- Angewandte Chemie International Edition
- Chemistry - A European Journal
- European Journal of Inorganic Chemistry
- European Journal of Organic Chemistry
- ChemSusChem
- Zeitschrift für anorganische und allgemeine Chemie

In case the system cannot determine the publisher (or the journal, if the publisher is Wiley) it will perform a generic crop that sometimes might crop too much. However, the user still has the option to disable the crop using the crop_pdf argument.

The final argument we offer is headers_to_ignore, which the user can take advantage of to discard certain sections. In our experiments, we identified sections that contain text that might not be too relevant such as the Bibliography or References. After instantiating the PDFReader, the user can use the ".headers" attribute to get a list of the headers the system identified. From there, the user can pass a list of all the headers they want the system to ignore when extracting the text. This will discard those headers and their corresponding sections from the extracted text.

After calling the extractText method, the user can access the text as a string variable or can access a dictionary where the keys correspond to the headers and the values correspond to the text that belongs to each header.

6.2 Batch File Reader

The batch file reader has a similar structure to the individual file reader. The system expects a constructor and then a call to extractText. However, each one of those works a bit differently.

The constructor for PDFBatchReader expects three arguments:

- pdf_folder_path
- text_folder_path
- imageless_folder_path

pdf_folder_path will contain the path to the folder where all the PDFs to convert are stored; text_folder_path will hold all the text files that correspond to the PDFs in the previous folder; imageless_folder_path will hold a copy of the PDFs without their images. We chose to store the imageless version of the PDFs in a separate folder because, as researchers, we understand how much time is allocated to identifying key papers and wouldn't want to destroy anybody's collection.

After initializing the instance and the constructor is called, users can see the headers of all the papers in the PDF path previously specified. Using this list, they can select the headers that they wish to discard using the extractText() method. Similar to the individual file processing, extractText() expects different arguments. Specifically, it expects a list of headers to ignore and a boolean to determine whether or not the output text should be formatted. The system takes advantage of the interface defined in the individual file reader to avoid having to redefine most methods.

A key limitation that we encountered was the lack of a Python package for TextWorks. Unfortunately, this library only offers a command line interface. However, Python offers a module, named subprocess, that allows to create new processes and obtain their return codes. We took advantage of this module and created a Python interface that spawns a TextWorks process to extract the text from the PDF.

Finally, the user is left with the folder specified in imageless_folder_path filled with the text versions of the PDFs.

6.3 Tokenization

After we automated the text collection part, we moved on to the next stage in the NLP pipeline: tokenization. This is "the process to identify tokens, or those basic units which need not be de-

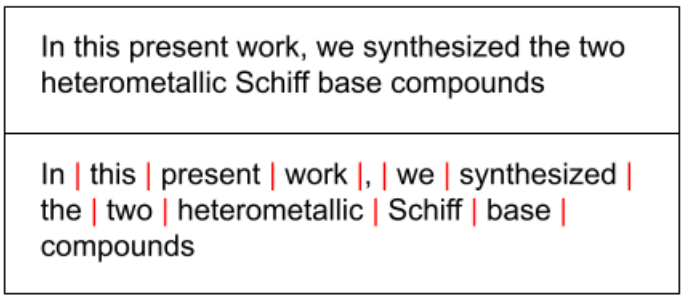

Figure 6.1: A sample of trivial tokenization.

composed in a subsequent processing" [15]. While sometimes this task is a trivial one, that is not always the case.

For example, look at Figure 6.1. It shows an example of tokenization. In this sentence, it is easy to determine what are the tokens as the red markers show. However, Figure 6.2 shows how tokenization can get a bit more complicated when we add more characters. In this example, we see how the addition of the dash character "-" and parentheses "(" and ")" can make us question what we define as tokens in the text. Both of the options in Figure 6.2 are valid, but we should be careful as to how this will affect the NLP pipeline in future steps.

While the tokenization problem has been tackled in the biomedical field (with work such as Bennet et al. [2]), it hasn't been explored enough in the catalyst field. However, there are plenty of libraries that offer their own versions of tokenization like ChemDataExtractor [14], Chemlistem [3], Standford's Stanza [11], and CoreNLP [7]. Since we had access to two PhD students with relevant experience in the field, we wanted to compare the performance of these different tokenizers in text that is relevant to our purposes. Therefore, Joseph Karnes and Emily Mikeska manually tokenized four research papers in their entirety. However, one of the issues that we encountered is the lack of a metric to measure tokenization accuracy. Furthermore, comparing tokenization is not such a trivial matter as Figure 6.3 shows.

A seemingly simple way to compare two lists of tokens would probably be to iterate over both lists at the same time and for each iteration add a score if the selected tokens are the same. This idea

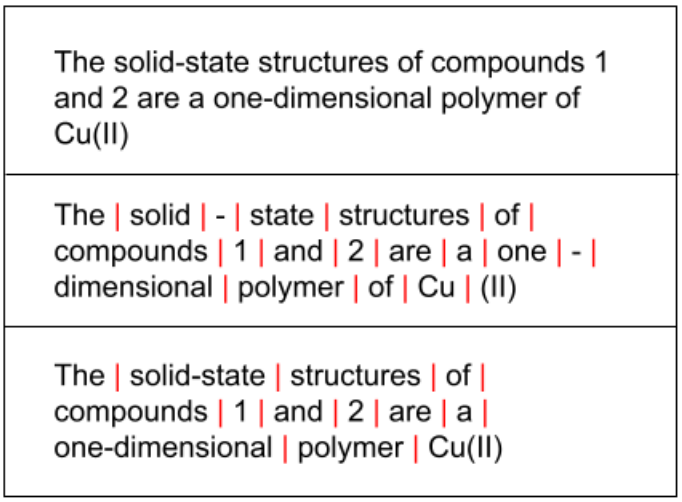

Figure 6.2: A sample of non-trivial tokenization.

has one key issue though: What happens after we encounter two mismatched tokens? Figure 6.3 shows a case where we compare two lists of tokens. On the second row, there is a token mismatch that cascades and causes every single future token to be considered a mismatch too. This situation would lead us to conclude that both lists only have one token in common. However, this is not an accurate representation of what is actually happening, as we can see that there are several tokens that match between both lists.

We formulated four tokenizer metrics that use different approaches to try to measure the similarity between two tokenizer lists. The first metric merely calculates a difference of the length of the list of tokens. The second one takes advantage of the set data structure in Python. It converts both lists into sets and then calculates the intersection between both of them. The third metric uses a concept known as Levenshtein distance. This distance between two words is determined by the number of characters that need to be modified/removed/added so that both words are the same [16]. Therefore, the idea for the third metric is to take advantage of the Levenshtein distance and estimate the Levenshtein distance between the tokens in each list, if this distance is higher than 0 then it adds the value to a score tracker.. For each unmatched token, it looks 15 tokens ahead

The	The
solid-state	solid
structures	state
of	structures
compounds	of
1	compounds
and	1
2	and
are	2
a	are
one-dimensional	a
polymer	one
of	dimensional
Cu(II)	polymer
	of
	Cu
	(II)

Figure 6.3: 1 on 1 match of different styles of tokenization for the same sentence.

iterating to the next one if this search fails. Five times we repeat this search until the algorithm enters a panic mode during which it searches for the next tokens that match. The algorithm uses this approach to avoid cascading one error down the lists. The final metric does not spend time looking ahead for tokens that might match, it jumps straight into panic mode. Initially, I did no want to go into panic mode too often to improve the performance of the metric in terms of execution time. However, I noticed that during my tests, execution time was not an issue at all, so I moved on to make the fourth metric.

Chapter 7

Results

As mentioned before, one of the packages that we chose to use (TextWorks) does not have a Python interface available [1]. The package was written in Scala; therefore, there is no native way to execute any of its modules from Python like it is possible with C++. However, TextWorks offers a command-line interface, so we decided to create a wrapper that makes the command-line interface calls. Due to our structure of processing individual files as well as batch processing whole folders, we created two classes: TextworksFileParser and TextworksBatchParser. Both classes have a constructor and a extractText() method. The extractText() method makes the call to TextWorks, which generates a text file in the same directory and our Python wrapper locates this file and imports its contents into memory.

The following code displays how we use the TextworksFileParser in our extractText() method within the PDFReader class. In line 3, we instantiate the variable by passing self.preprocessedpdf, the path to the pre-processed PDF, as an argument. In line 4, we just need to call the extractText() method that internally will make the call to TextWorks, and then it will locate the file it generates and import its contents into memory, which we then assign to self.textworks_json. This way, we managed to solve the issue of lacking a native interface for TextWorks in Python.

```
1 ...
2 if self.textworks_json is None:
3             textworks_parser = TextworksFileParser(self.preprocessedpdf)
4             self.textworks_json = textworks_parser.extractText()
5 ...
```

Identifying headers also proved to be a complicated problem to solve. Even though our idea

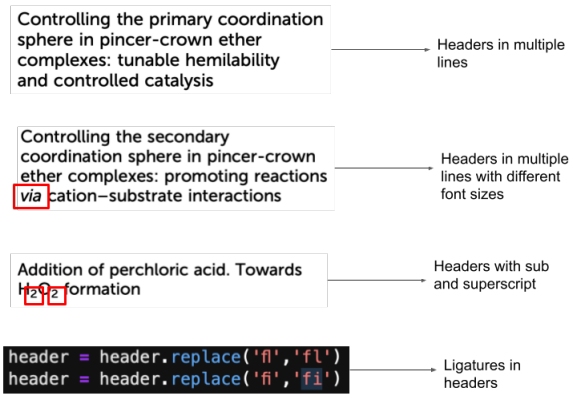

Figure 7.1: Different scenarios that our systems handles.

to search for common headers (Acknowledgments and References) worked, headers themselves had plenty of peculiarities within them. One of the first ones that we encountered was the use of different font-sizes within the same header. Then, we had to deal with headers that span more than one line. Finally, we dealt with headers that had sub- and super-scripts in them. After solving these problems, we had an accurate way of identifying complex headers within PDFs. Figure 7.1 shows an example of the different cases that we can handle.

Let's look at the different scenarios that our users can handle with pyCatalstReader.

```
1 from PDFReader import PDFReader
2
3 pdf_r = PDFReader('sample.pdf')
4 pdf_r.headers
5 pdf_r.extractText(remove_images=True, header_formatted=True,
6                    crop_pdf=True, headers_to_ignore= ['ACKNOWLEDGMENTS'])
7 pdf_r.saveToText()
```

On line 1, we import the package and the respective module. This case shows the easy functionality of the single file reader, so we import PDFReader. On line 3, we instantiate the variable by using the PDfReader constructor. As explained before, the constructor expects the path to a single PDF file, so in this case we use "sample.pdf". In line 4, the user takes advantage of our feature to extract headers by using the .headers attribute. In the next line, the user calls the extractText()

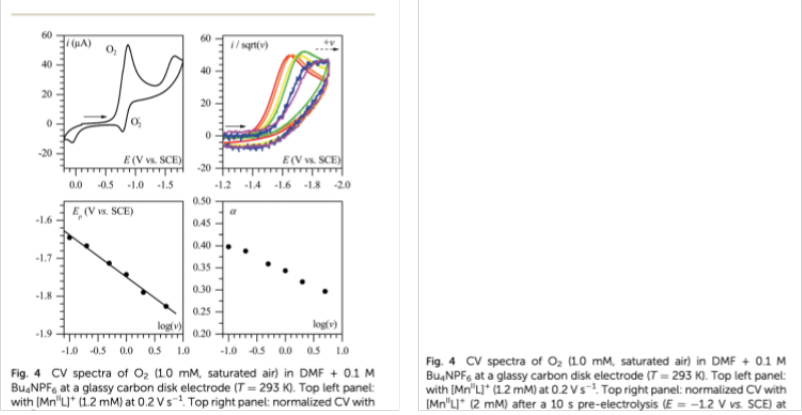

Fig. 4 CV spectra of O_2 (1.0 mM, saturated air) in DMF + 0.1 M Bu_4NPF_6 at a glassy carbon disk electrode ($T = 293$ K). Top left panel: with $[Mn^{II}L]^+$ (1.2 mM) at 0.2 V s^{-1}. Top right panel: normalized CV with

Fig. 4 CV spectra of O_2 (1.0 mM, saturated air) in DMF + 0.1 M Bu_4NPF_6 at a glassy carbon disk electrode ($T = 293$ K). Top left panel: with $[Mn^{II}L]^+$ (1.2 mM) at 0.2 V s^{-1}. Top right panel: normalized CV with $[Mn^{II}L]^+$ (2 mM) after a 10 s pre-electrolysis ($E = -1.2$ V *vs.* SCE) at

Figure 7.2: On the left, the original PDF. On the right, the same section after we extract all images.

function along with some parameters. The first parameter (remove_images) makes sure that we are deleting all the images in the pdf. Figure 7.2 shows an example of a PDF after all images were deleted. In this example, if the image was kept in the PDF, our package would have identified all the text inside the graphic, such as quantities in the axis. The next one (header_formatted) makes sure that the extracted text is formatted around the headers. The third parameter, crop_pdf, crops the margins that might carry some useless text. The final parameter, headers_to_ignore, specifies the section headers that the user wants to discard during the extraction process.

After this step, users have two options: they can either choose to use the text in the same script by using the .text attribute or save the text to a file by calling the saveToText() function. In line 6, we chose the latter option and called the saveToText() function. With its default parameters, this function will append the suffix "_text" to the original filename. Therefore, after the whole process, we are left with two new files: sample_Imageless.pdf and sample_text.pdf, while sample.pdf remains untouched. Figure 7.3 shows an example on how our system is capable of capturing headers as well as super and subscript. This extraction was performed using the header_formatted parameter set to True in the extractText() method. Therefore, the system will automatically split the text based on its headers.

Now let's move on to the batch file reader.

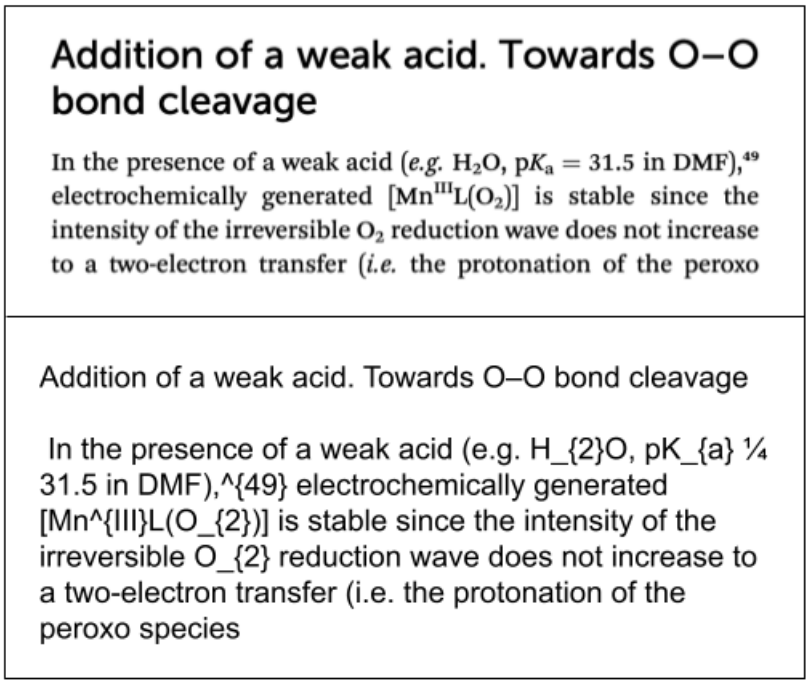

Figure 7.3: On top we see a portion of the PDF file, on bottom how we capture the text.

```
1 from PDFReader import PDFBatchReader
2
3 batch_pdf = PDFBatchReader(pdf_folder_path='/pdf_folder',
    imageless_folder_path='/pdf_noimages_folder', text_folder_path='/
    pdf_text_folder')
4 batch_pdf.headers
5 batch_pdf.extractText(headers_to_ignore=['Acknowledgments', '
    Acknowledgements','Acknowledgments & References'])
```

The first line is different from the individual file reader example. In this case, since we want to use the batch file reader, we import PDFBatchReader. However, after that we can see some similarities with the individual file reader. In line 3, we instantiate the variable by calling the constructor to PDFBatchReader. However, in this case we pass a folder as the first argument, and we do the same for all the other arguments. The first argument (imageless_folder_path) specifies the folder that contains all the PDF files. The second argument (imageless_folder_path) specifies

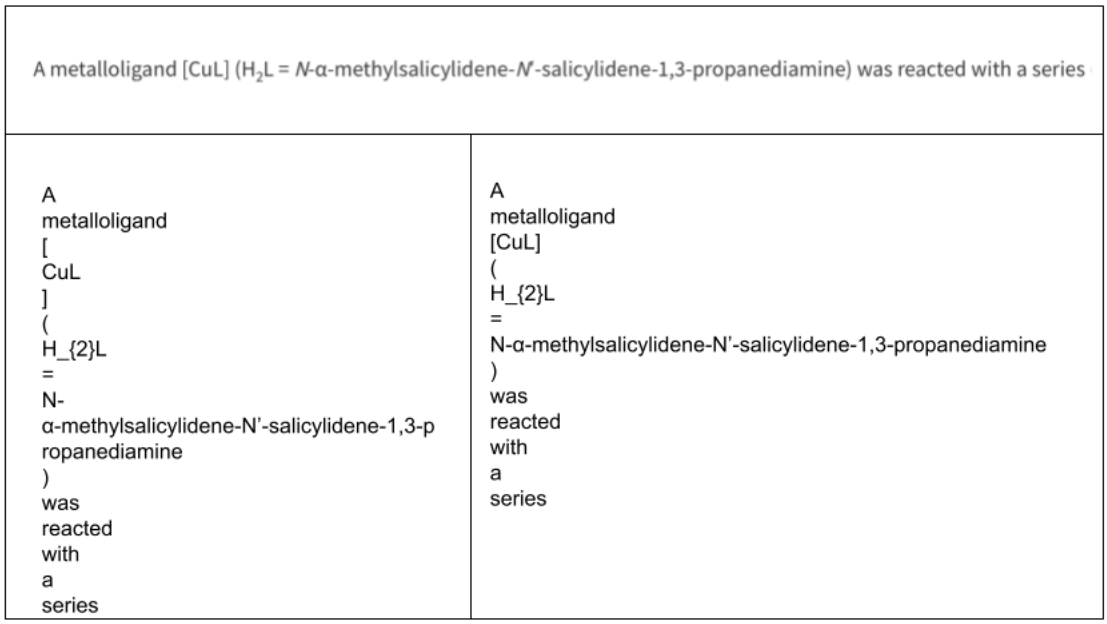

Figure 7.4: Comparison of traditional tokenization vs chemistry-oriented tokenization.

where to store the imageless versions of the PDFs. Finally, the last argument (text_folder_path) specifies where to store the files containing the PDF text.

After this step, users can visualize the headers of all the PDFs using the headers attribute. In this list, they can identify headers that they want to discard during the text extraction process. In line 5, we called the extractText() function with a list of headers as the parameter for headers_to_ignore. A key difference in this case is that extractText() automatically stores the files containing the extracted text from the PDFs in the folder specified in the text_folder_path parameter in the constructor.

Moving on to the tokenization metrics, here are the comparison graphs that we obtained after running the four different metrics in the papers that Joe and Emily manually tagged. Figures 7.6 and 7.7 show the two more advanced tokenization metrics. I have decided not to display the other two as they were too basic and did not capture any major differences between the different tokenizers that we tested. As mentioned before, tokenization is a problem that has been thoroughly explored for daily language, but the field of Chemistry still remains as a big challenge due to its usage of the dash "-" character.

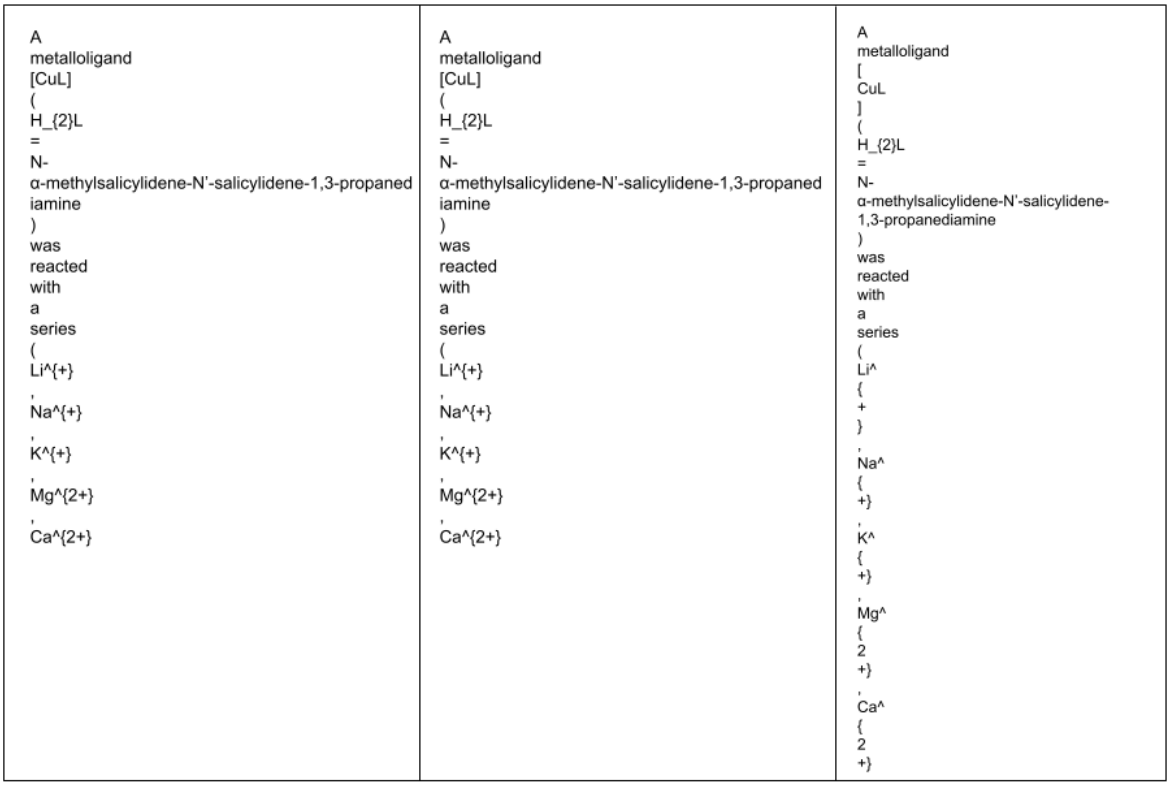

Figure 7.5: Comparison between expert tokenization and ChemDataExtractor's and Stanza's.

Figure 7.4 shows a comparison of the tokenization approaches between Stanza and ChemDataExtractor. At the top of the graphic, we have the text as it shows up on the PDF document. At the bottom right and left, we can see the text after it is extracted with our package and tokenized using the two methods. On the bottom left we can see the text after it is tokenized using Stanza's tokenizer and on the bottom right after it goes through ChemDataExtractor's tokenized. As explained before, Stanza's tokenizer is oriented towards every day language while ChemDataExtractor's tokenizer is more oriented towards chemistry related text. From the example in Figure 7.4, we can see the different ways that both packages handle the dash character "-". While Stanza seems to split more easily at the dash character, ChemDataExtractor does a better job at keeping the whole chemical entity as a single token.

For these experiments, Joe and Emily tokenized four research papers in the catalyst field. Since

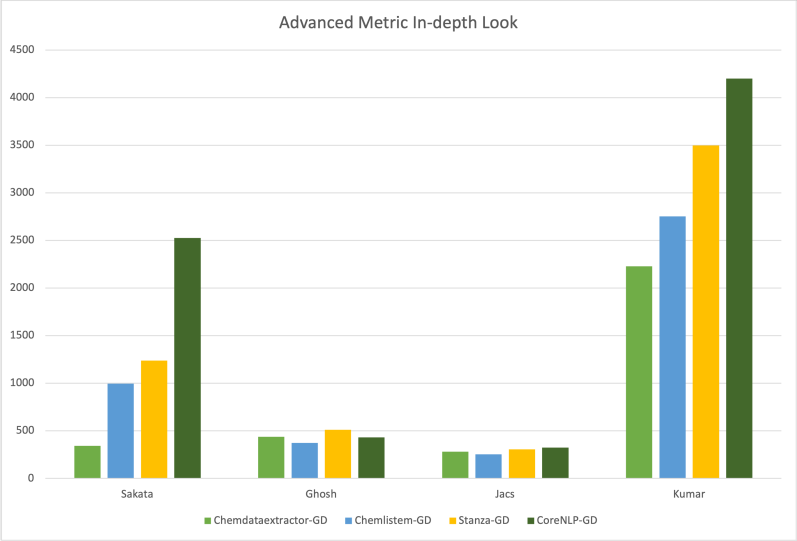

Figure 7.6: Metric that takes advantage of Levenshtein distance, on the x-axis the performance of the different tokenizers on each document. The lower the score, the better the tokenizer performs.

tokenization depends a lot on the context where it is used, we wanted to take advantage of their expertise in the field and compare their manual tokenization with automated tokenization options. The five tokenizers that we tested were: ChemDataExtractor, Chemlistem, Stanza, CoreNLP, and a tokenizer created by Andrew Jenny of Team CO2. Both Stanza and CoreNLP are tokenizers created by the Stanford NLP group that are used for everyday language. However, CoreNLP is older so it is a bit more stable and can lead to better results, so we decided to include both of them just in case. Figure 7.5 shows a comparison between the manual tokenization in the left column, ChemDataExtractor's tokenization on the middle and Stanza's on the right. We can see how ChemDataExtractor's results closely match the manually tokenized text. In particular, it is interesting to see the different ways that Stanza and ChemDataExtractor handle characters such as the dash "-" or the curled brackets "{". This is just one example of how chemistry-oriented tokenizers should not follow the same rules as regular tokenizers.

We created graphs to compare the performance of the tokenizers in the four papers that Emily and Joe tagged. The metrics that we developed showed the distance between two list of tokens; therefore, the lower the score, the better the performance of the tokenizer. For our experiments we calculated the distance between the manually extracted tokens and the ones using the automatic

tokenizers. For these graphics, we are comparing the more advanced metrics that use the Levenshtein distance. Figure 7.6 shows the distance calculated using the metric that uses the lookahead value of 15. Along the x-axis, we have the the four different papers that Emily and Joe tagged: Sakata, Ghosh, Jacs, and Kumar. On the y-axis, we have the distance value for each one of these papers. While the values on Ghosh and Jacs do not seem to show any major difference between the tokenizer, the story is different for Sakata and Kumar, where we can see a clear advantage for both ChemDataExtractor and Chemlistem. From a visual analysis, we can see why as the same situation that we saw in Figure 7.5 occurs, where Stanza could not capture the right patterns in the language used in the field of chemistry.

Figure 7.7 includes the tokenizer that Andrew Jenny developed focused on the field of chemistry, but aimed towards another specific topic. The metric we use in this case is the deep metric that does not use any lookahead value and therefore performs a more in-depth comparison between two lists of tokens. This metric shows how Andrew's tokenizer can perform much better than Chemdataextractor in some instances like in Sakata and Jacs. We followed this by doing a more manual analysis of the tokens that both systems returned. During this analysis, we noticed that both algorithms performed quite similarly, but we did notice a big difference as to how they deal with the dot "." that finishes sentences. While Andrew's tokenizer used a series of rules to perform this split, ChemDataExtractor uses the Punkt Algorithm. This algorithm is an "unsupervised approach to sentence boundary segmentation" [5]. It trains its model on a set of documents to accurately guess when the dot character is actually ending a sentence. We found that the Punkt algorithm's sentence boundary detection works particularly well in chemistry-related text.

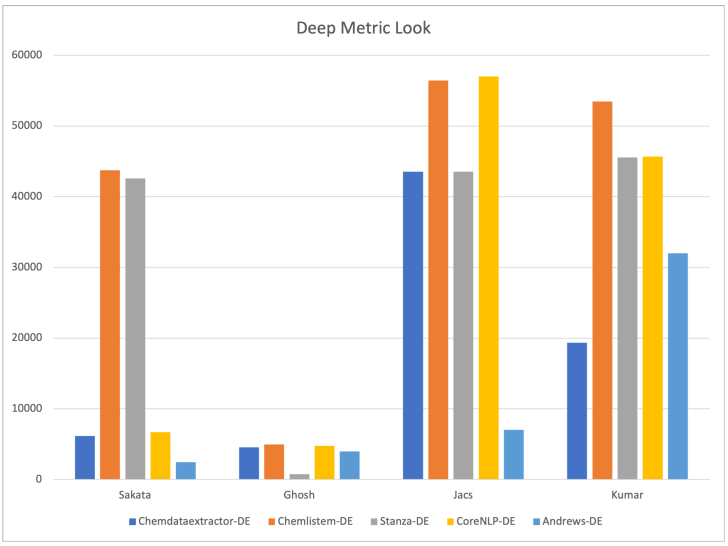

Figure 7.7: Deep metric that does not use any parameters to estimate the estimate the tokenization performance. We also include Andrew Jenny's tokenizer in this analysis.

Chapter 8

Conclusion

In this report, we have discussed the importance of developing a tool that can extract text out of catalyst science research papers. NLP is a potential solution that could help accelerate the discovery of new catalysts that will help society. However, without any way to automate the collection of a valid corpus in this field, it will be impossible to test any of these theories. While trying to explore NER, an NLP tool, we identified the lack of a system that could deal with the peculiarities of research papers in this field such as the usage of super- and sub-scripts, two column formatted text, discarding certain sections in the resulting text, cropping the margins of the file, and removing any graphics. Therefore, we proceeded to explore the possibility of developing our own tool that could deal appropriately with these constraints.

Using the field knowledge of Emily and Joe, we identified PDFs as the most popular format to distribute the papers we wanted to analyze. This posed several issues as PDF files are built to facilitate their visualization on multiple mediums, not their analysis. While HTML files define a clear structure through its usage of tags, PDFs contain little metadata that could help define a structure within the file. However, as stated before, PDFs are the most popular format to share the files we were interested in, so we had no other option but to navigate through the obstacles that PDFs carry.

During the beginning stages of this project, we explored several open-source packages with the hope that one of them could cover all the scenarios previously mentioned. What we encountered were packages that were good at one or two of our requirements. In some cases, we couldn't find a package capable of doing what we wanted at all. Therefore, we spent the majority of this project developing a package that integrated the functionality of different tools. Specifically we

used PyMuPDF for font information capture and identifying images, PyPDF2 to remove images and TextWorks to extract the final text respecting sub- and super-script characters. Due to the interdisciplinary nature of our team, we had to split responsibilities accordingly. I, as our only member with a Computer Science background, served as the architect as well as lead developer in our team. The rest of our team went through a learning phase during which they learned key programming concepts as well helped define the requirements of the package we envisioned back then. We decided to name this tool pyCatalstReader.

Specifically pyCatalstReader has the following abilities:

- Identify headers within a PDF and split the text into its sections.
- Using the header information, it can remove certain sections to avoid adding unnecessary data into the resulting text.
- Remove images that contain text that without the visual context might not make much sense.
- Appropriately extract text taking into consideration the double-column format as well as sub- and super-scripts.
- Crop the margins of PDFs that belong to a predefined list of publishers and journals.
- Extract the text from a single PDF or a folder that contains several PDFs without modifying the original file.

Once we had a way to automate the extraction of text from PDFs, we moved on to the first step of the NLP pipeline: tokenization. Specifically, in chemistry-related text, tokenization has not been explored as much as in everyday language. From the beginning, we theorized that tokenizers optimized for regular language could not perform as well in chemistry-related text. Using pyCatalstReader, we extracted the text and Emily and Joe manually tokenized the text. However, we identified the lack of an objective measure to measure tokenization accuracy. Therefore, we developed metrics that could capture this information. We designed four metrics, but only the last two showed any meaningful data. These last two metrics take advantage of the Levenshtein

distance, which measures the number of characters that have to be added/removed/modified for two words to become the same. These last two metrics showed ChemDataExtractor, a chemical NER package, as the clear winner. We also analyzed the performance of the tokenizer created by Andrew Jenny, a member of another team in the same grant. His tokenizer performed remarkably well, even outperforming ChemDataExtractor in some instances. However, after close analysis we noticed that ChemDataExtractor's usage of the Punkt algorithm allows it to split sentences more accurately.

NLP techniques in the field of chemistry have not been explored as thoroughly as in other fields due to the lack of an available corpus. Most of the models built to perform chemical NER have to be built using a corpus extracted from patents as well as abstracts. This is because these two categories of text are the only ones available programatically. We hope that pyCatalstReader can help the community collect a larger corpus of text relevant to their studies and explore NLP techniques that could lead to the next big discovery. Furthermore, we also compared different tokenizers to objectively identify the one that performs that best for the task relevant to our research group. We hope our findings can help future cohorts in our team.

While we hope to have made a meaningful contribution to future NLP research in the field of chemistry, there is still more work that could be done. One of the missing key features is the ability to parse tables within the PDF files. Usually tables contain structured information that would be helpful to extract. Similarly, it would be interesting to extract information out of the graphics in research papers. Currently, by default, we remove any image in the PDF files to avoid extracting text from the axis. However, graphics are a rich source of information that complements the text in the research paper.

On the tokenizer side, while we recommend using ChemDataExtractor's tokenizer in the chemistry text, it would be interesting to keep exploring Andrew Jenny's tokenizer. ChemDataExtractor uses the Punkt algorithm to perform sentence segmentation and it could complement Andrew's tokenizer. Also, it would be interesting to explore Deep Learning options to perform the tokenization and text extraction steps.

References

[1] A. N. S. Adam Saunders. Watr-Works, 2020. Available electronically at: `https://github.com/iesl/watr-works`.

[2] N. A. Bennett, Q. He, K. Powell, and B. R. Schatz. Extracting Noun Phrases for all of MEDLINE. In *Proceedings of the AMIA Symposium*, page 671. American Medical Informatics Association, 1999.

[3] P. Corbett and J. Boyle. Chemlistem: Chemical Named Entity Recognition Using Recurrent Neural Networks. *Journal of Cheminformatics*, 10(1):1–9, 2018.

[4] Hazy Research. pdftotree, 2020. Available electronically at: `https://github.com/HazyResearch/pdftotree`.

[5] T. Kiss and J. Strunk. Unsupervised Multilingual Sentence Boundary Detection. *Computational Linguistics*, 32(4):485–525, 2006.

[6] E. D. Liddy. Natural Language Processing. In *Encyclopedia of Library and Information Science, 2nd Ed.*, 2001.

[7] C. D. Manning, M. Surdeanu, J. Bauer, J. R. Finkel, S. Bethard, and D. McClosky. The Stanford CoreNLP Natural Language Processing Toolkit. In *Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 55–60, 2014.

[8] J. X. McKie. PyMuPDF, 2021. Available electronically at: `https://github.com/pymupdf/PyMuPDF`.

[9] V. Mehta. poppler-utils, 2020. Available electronically at: https://pypi.org/project/poppler-utils/.

[10] National Science Foundation. NRT-HDR: Internet of Catalysis - Harnessing Data Science for Catalyst Design, 2019. Available electronicaly at: https://www.nsf.gov/awardsearch/showAward?AWD_ID=1922649&HistoricalAwards=false.

[11] P. Qi, Y. Zhang, Y. Zhang, J. Bolton, and C. D. Manning. Stanza: A Python Natural Language Processing Toolkit for Many Human Languages. *arXiv preprint arXiv:2003.07082*, 2020.

[12] Y. Shinyama. pdfminer, 2019. Available electronically at: https://pypi.org/project/pdfminer/.

[13] J. H. Sinfelt. Bimetallic Catalysts. *Scientific American*, 253(3):90–101, 1985.

[14] M. C. Swain and J. M. Cole. Chemdataextractor: A Toolkit for Automated Extraction of Chemical Information from the Scientific Literature. *Journal of Chemical Information and Modeling*, 56(10):1894–1904, 2016.

[15] J. J. Webster and C. Kit. Tokenization as the Initial Phase in NLP. In *COLING 1992 Volume 4: The 14th International Conference on Computational Linguistics*, 1992.

[16] L. Yujian and L. Bo. A normalized Levenshtein Distance Metric. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 29(6):1091–1095, 2007.